Security Review Report
NM-0394 EtherFi Cashback

NETHERMIND
SECURITY

(Dec 13, 2024)

Contents

1 Executive Summary

This document presents the security review performed by Nethermind Security for the EtherFi Cash contracts. This security engagement focused on the smart contracts that constitute part of the new payment system developed by the **EtherFi** team. This update introduces a **cashback** feature into the **EtherFi Cash Contracts** system as well as quality of life improvements. The audit was split into two parts. The first part introduced the new Cashback system, and after the first review and first round of fixes, the code was updated and it was audited again. This report presents the findings identified across both of these sessions. The audit started with the commit 26bce2a and was expanded with new features at the commit 5052bb7.

The audit was performed using (a) manual analysis of the codebase, (b) automated analysis tools, (c) simulation of the smart contract, and (d) creation of test cases. **Along this document, we report** nineteen points of attention, where one is classified as `High`, one is classified as `Medium`, seven are classified as `Low` and twelve are classified as `Informational` or `Best Practice`. The issues are summarized in Fig. 1.

This document is organized as follows. Section 2 presents the files in the scope. Section 3 summarizes the issues. Section 4 presents the system overview. Section 5 discusses the risk rating methodology. Section 6 details the issues. Section 7 discusses the documentation provided by the client for this audit. Section 8 concludes the document.

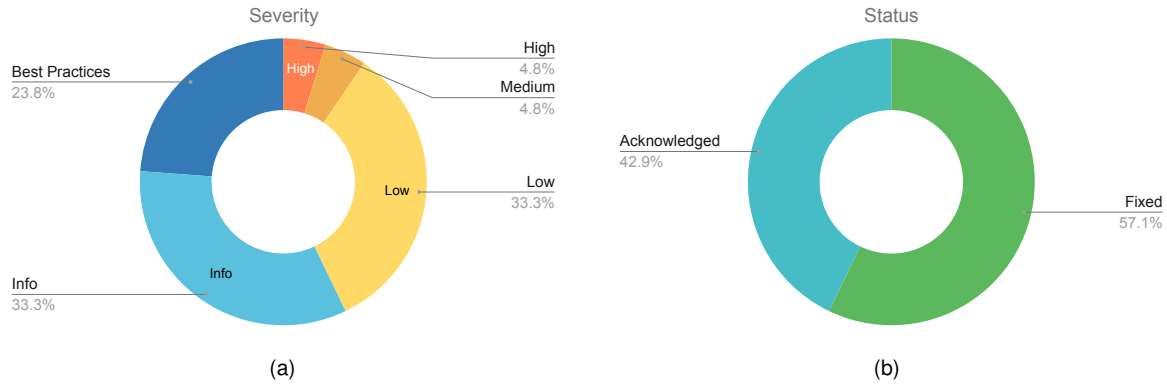

Fig. 1: Distribution of issues: Critical (0), **High** (1), **Medium** (1), **Low** (7), **Undetermined** (0), **Informational** (7), **Best Practices** (5). **Distribution of status: Fixed** (12), **Acknowledged** (9), **Mitigated** (0), **Unresolved** (1)

Summary of the Audit

Audit Type	Security Review
Initial Report	December 13, 2024
Response from Client	Regular responses during audit engagement
Final Report	December 13, 2024
Repository	cash-contracts
Commit (Audit)	26bce2a3c5f92ffc4aec20c6d9a06d32745d1be7
Commit (Additional Feature)	5052bb7a1747a52842ba6ae1fb3a3439b04e1a0a
Commit (Final)	3e8bcb4980388e255d05deb1cd2fe3f5958f4ee5
Documentation Assessment	Low
Test Suite Assessment	Medium

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	cashback-dispatcher/CashbackDispatcher.sol	110	5	4.5%	29	144
2	user-safe/UserSafeSetters.sol	235	3	1.3%	40	278
3	user-safe/UserSafeStorage.sol	240	17	7.1%	39	296
4	user-safe/UserSafeEventEmitter.sol	113	1	0.9%	30	144
5	user-safe/UserSafeLens.sol	75	1	1.3%	17	93
6	user-safe/UserSafeCore.sol	302	16	5.3%	63	381
7	user-safe/UserSafeFactory.sol	76	11	14.5%	16	103
8	debt-manager/DebtManagerAdmin.sol	125	1	0.8%	34	160
9	debt-manager/DebtManagerCore.sol	498	27	5.4%	114	639
10	debt-manager/DebtManagerStorage.sol	308	20	6.5%	40	368
11	debt-manager/DebtManagerInitializer.sol	14	4	28.6%	4	22
12	interfaces/IUserSafe.sol	136	185	136.0%	38	359
13	interfaces/IL2DebtManager.sol	243	234	96.3%	52	529
	Total	**2475**	**525**	**21.2%**	**516**	**3516**

3 Summary of Issues

	Finding	Severity	Update
1	Users can block liquidations because swaps and repay don't update withdrawal request	High	Fixed
2	Lack of collateral cap	Medium	Acknowledged
3	Inconsistent updates of mode on swap	Low	Acknowledged
4	Lack of partial liquidation	Low	Fixed
5	Liquidators can steal user's asset that is not part of the listed collateral tokens	Low	Fixed
6	CashbackDispatcher::setCashbackToken‘ allows setting cashback tokens that don't have the price configured yet	Low	Fixed
7	Removal of the supported collateral may break the invariant	Low	Fixed
8	Unsupporting a Borrow Token can be prevented through front-running	Low	Acknowledged
9	Incorrect information from canSpend(...) and maxCanSpend(...) in UserSafe	Low	Acknowledged
10	Incorrect interface	Info	Fixed
11	Unnecessary updates of mode repay(...) and swapAndRepay(...)	Info	Fixed
12	Unused internal function _checkSpendingLimit(...)	Info	Fixed
13	Automatic switching to credit mode can return wrong possible expenditure for a user	Info	Acknowledged
14	The monthly spending limit is limited by the daily spending limit	Info	Acknowledged
15	UserSafeStorage::swapFunds will revert when working with stETH and eETH	Info	Acknowledged
16	Credit Mode will not start at _incomingCreditModeStartTime	Info	Acknowledged
17	Add check if the token is supported on repay(...)	Best Practice	Fixed
18	Wrong code comments	Best Practice	Fixed
19	Inconsistent use of ‘ADMIN_ROLE‘ and ‘DEFAULT_ADMIN_ROLE	Best Practice	Acknowledged
20	Lack of address(0) check for recipient param inside requestWithdrawal function	Best Practices	Fixed
21	No cap for the borrowApy	Best Practices	Fixed

4 System Overview

The main entry point for the KYC'ed users is `UserSafe`, to which users deposit on-chain funds and with which users interact to manage funds. The `UserSafe` can transfer funds to the multisig wallet controlled by EtherFi, which is connected to the off-chain card payment infrastructure. The `UserSafe` contracts are created from a factory contract `UserSafeFactory`. `UserSafe` can perform swaps with external protocols and use `L2DebtManager` to take over-collateralized loans. Those funds are used to pay with a card. The `L2DebtManager` calculates the value of collateral by calling `PriceProvider`, which fetches the price from the `Chainlink Oracle`. The USD stablecoins comprise the borrowing pool in `L2DebtManager`. Those funds can be supplied by the non-KYC'ed participants who get shares in exchange. Those shares accrue in value over time by increasing the borrowers' loan value. The unhealthy loans can be liquidated by non-KYC liquidators who are incentivized by the liquidation bonus. The latest update also included a **cashback mechanism** that allows EtherFi Cash card users to earn rewards for using the card in the form of cashback. There are 5 cashback tiers and depending on the tier that the users are part of, their cashback percentage will adjust accordingly. The higher the tier, the higher the cashback percentage will be.

The following sections delve into the system's components and their interactions. The diagram below showcases a high-level view of the system's architecture.

4.1 Protocol participants

The `EtherFi Cash` has following participants:

- **Users** - the protocol users that went through the KYC (Know Your Customer) process and are eligible to create `UserSafe` contract, which can be utilized in the card payment process. Users are the owners of the `UserSafe`
- **EtherFi Cash Wallet** - special address controlled by the EtherFi. From this address, users can call their `UserSafe` contracts to manage funds
- **EtherFi Cash Multisig** - special address controlled by the EtherFi, which receives funds on transfers and borrows. Funds in this address can be used to perform card payments
- **Contracts Owner** - special addresses controlled by the EtherFi that control the `L2DebtManager` and `UserSafeFactory` contracts
- **Borrow tokens suppliers** - any address can supply USD stablecoins to the `L2DebtManager` in exchange of shares
- **Liquidators** - any address can perform liquidations of the liquidatable loans

4.2 UserSafe - funds management

The users, after the KYC process, are eligible to register the `UserSafe` contract so that funds from this contract can be used to spend with a `EtherFi Card`. The user who is the owner of the `UserSafe` contract can perform configuration actions on the `UserSafe` contract with the following functions:

- `setOwner(...)` - checks the provided signature and sets new owner for the `UserSafe` contract
- `setMode(...)` - allows users to switch between Debit and Credit modes
- `updateSpendingLimit(...)` - changes spending limit while preserving the spent amount
- `requestWithdrawal(...)` - creates a request withdrawal which defines assets and amounts that will be withdrawn after the predefined period.
- `processWithdrawal(...)` - permisionless function that finalizes the withdrawal request process
- `setUserRecoverySigner(...)` - allows users to set a recovery signer
- `recoverUserSafe(...)` - a permissionless function that requires two valid signatures from predefined recovery addresses. This function allows for emergency recovery of the `UserSafe` account by changing the current owner to the new owner.

Users are able to move the funds in/from their `UserSafe` by utilizing the following functions:

- `swap(...)` - users can swap from a provided input token to a desired output token. The function will also check that a UserSafe borrowings don't exceed the expected ltv.
- `canSpend(...) and maxCanSpend(...)` - view functions that will check if the user can spend the desired amount depending on the current mode.
- `spend(...)` - allows users to spend the funds from their UserSafe
- `swapAndSpend(...)` - convenience function that allows users to swap and then spend in a single transaction
- `repay(...)` - allows users to repay their loans
- `swapAndRepay(...)` - convenience function that allows users to swap and repay their loans in one transaction
- `retrievePendingCashback(...)` - allows users to claim the cashback rewards that they are entitled to.

4.3 L2DebtManager - loans mechanism

The loan mechanism allows users to borrow supported USD stablecoins pooled in the `L2DebManager`. Total debt and total collateral are tracked for each user, so it is not possible for a single user to create more than one loan. To borrow a stablecoin, the user needs to provide enough collateral since the loans are overcollateralized. The amount of collateral is stored in native value and transformed to value in USD (by current price) when needed. The collateral may be withdrawn only if the state of the loan remains healthy after the withdrawal. The health of a loan is defined for each token in the `_ltv` mapping, which represents the maximum proportion of the debt to collateral under which the loan is healthy. The borrowing tokens can be supplied by any address in exchange for shares. Shares accrue in value over time, which incentivizes the suppliers to provide liquidity to the `L2DebtManager`. The borrowers are obligated to repay a higher amount of debt, which constitutes the supplier's rewards. The loan can become a liquidatable state, which is defined for each token in the `_liquidationThreshold` mapping, which represents the proportion of the debt to collateral above which the loan is considered liquidatable. Liquidation is permissionless, and liquidators are incentivized by the reward in the form of an additional collateral bonus defined for each token.

Below, we list core user-facing functions of `L2DebManager`:

- `supply(...)` - transfers the stablecoin from the caller, calculates the amount shares, which represent the portion of all the amount of given tokens and stores the number of shares
- `withdrawBorrowToken(...)` - calculates the number of shares that need to be removed for a given token amount, updates shares, and transfers the stablecoin to the caller
- `borrow(...)` - updates user's debt, checks loan's health, and transfers the stablecoin to the caller
- `repay(...)` - updates user's debt and transfers funds from the user
- `liquidate(...)` - checks if the loan is liquidatable, transfers stablecoin from the user, calculates the amount of the collateral needed (with liquidation bonus), updates user's debt and collateral, sends the collateral to the liquidator

4.4 CashbackDispatcher - cashback mechanism

This contract dispatches cashback to EtherFi cash users. The cashback mechanism allows users to earn cashback rewards depending on their current `UserSafeTiers`. There are 5 different tiers, each allowing users to earn a higher percentage of cashback rewards.

Below, we list the core functions of the `CashbackDispatcher` contract, excluding setter and admin function:

- `cashback(...)` - allows a `UserSafe` to claim the cashback rewards that it is entitled to while spending considering that the `CashbackDispatcher` has enough tokens. In case the `CashbackDispatcher` doesn't have enough tokens, the UserSafe will update the `_pendingCashbackInUsd` rewards so that the rewards will be claimable at a later date.
- `clearPendingCashback(...)` - allows a `UserSafe` to claim the cashback rewards that it is entitled to in case these were not automatically claimed while spending.

5 Risk Rating Methodology

The risk rating methodology used by Nethermind Security follows the principles established by the OWASP Foundation. The severity of each finding is determined by two factors: **Likelihood** and **Impact**.

Likelihood measures how likely the finding is to be uncovered and exploited by an attacker. This factor will be one of the following values:

a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;

b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;

c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding, other factors are also considered. These can include but are not limited to motive, opportunity, exploit accessibility, ease of discovery, and ease of exploit.

Impact is a measure of the damage that may be caused if an attacker exploits the finding. This factor will be one of the following values:

a) **High**: The issue can cause significant damage, such as loss of funds or the protocol entering an unrecoverable state;

b) **Medium**: The issue can cause moderate damage, such as impacts that only affect a small group of users or only a particular part of the protocol;

c) **Low**: The issue can cause little to no damage, such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding, other factors are also considered. These can include but are not limited to Data/state integrity, loss of availability, financial loss, and reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

		Severity Risk		
Impact	**High**	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Info/Best Practices	Low	Medium
	Undetermined	Undetermined	Undetermined	Undetermined
		Low	**Medium**	**High**
		Likelihood		

To address issues that do not fit a High/Medium/Low severity, Nethermind Security also uses three more finding severities: **Informational**, **Best Practices**, and **Undetermined**.

a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to pass to the client formally;

b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;

c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

6 Issues

6.0.1 [High] Users can block liquidations because swaps and repay don't update withdrawal request

File(s): `UserSafeCore.sol`, `UserSafeSetters.sol`

Description: During the call to functions `repay(...)`, `swap(...)` and `swapAndRepay(...)` the funds are moved out of the `UserSafe` but the amount in the pending withdrawal request is not updated.

The `UserSafe` can block liquidations, which would allow users to never repay a loan. Because the pending withdrawal request amount is not updated during swaps and repays this will lead to a situation where the balance of a `UserSafe` is < `_pendingWithdrawalRequest.amounts`. If this happens, then the `getUserTotalCollateral(...)` function, which is used during liquidation, would always revert due to an underflow error:

```
function getUserTotalCollateral() public view returns (IL2DebtManager.TokenData[] memory) {
//..
//..
    for (uint256 i = 0; i < len; ) {
        uint256 balance = IERC20(collateralTokens[i]).balanceOf(address(this));
        uint256 pendingWithdrawalAmount = getPendingWithdrawalAmount(collateralTokens[i]);
        if (balance != 0) {
        ////////////////////////////////////////
        // @audit: if the balance is lower than pendingWithdrawalAmount
        //         the call to function would always revert
        ////////////////////////////////////////
            balance = balance - pendingWithdrawalAmount;
            tokenAmounts[m] = IL2DebtManager.TokenData({token: collateralTokens[i], amount: balance});
            unchecked {
                ++m;
            }
        }
//..
//..
}
```

Below we present a possible scenario that would result in such an issue:

- assume the user already has some healthy 'loan A' in the volatile token but not in ETH

- user deposits to UserSafe 1 ETH

- user creates a withdrawal request of 1 ETH

- user borrows 1 ETH from the DebtManager (note that those funds are immediately transferred to the card, not to UserSafe)

- user repays 0.99 ETH of debt from the UserSafe balance

- now, if the loan A becomes unhealthy due to the borrowed token value increase during liquidation, the call to getUserTotalCollateral(...) would fail since the balance of ETH is 0.001 ETH but the withdrawal amount is 1 ETH and the 0.001 - 1 would result in revert.

Note that there is potential fund recovery from such a scenario that can be done by the DebtManager owner. Also, the users are KYC-ed, so the likelihood of such an attack is lower. Additionally such scenario could happend not as an attack but unintentionally which would block user from borrowing, spending and swapping.

Recommendation(s): Consdier updating the withdrawal amount when the funds are leaving UserSafe during repayment.

Status: Fixed

Update from client: e55b1b5e0cb5f3d00a3844c773c3df3dd537e14b

6.0.2 [Medium] Lack of collateral cap

File(s):

Description: Currently, there is no maximum amount of the collateral token that is backing the loans from `DebtManager`. This is unsafe because if a single asset is used in high volumes for the loans, a drop in the price may cause many loans to enter into an unhealthy state. Further liquidations would result in the immediate selling of the collateral asset, resulting in the collateral token price dropping even more, possibly causing new liquidations, causing the effect known as cascading liquidations.

Recommendation(s): Since the collateral is not deposited to `DebtManager` but stored in the `UserSafe`, the sum of all the collateral can't be easily tracked. One possible solution could involve tracking the number of registered `UserSafes` accounts and distributing the limit between all the `UserSafe` accounts. Example: if the limit for collateral in ETH is `10e7` and the number of all registered `UserSafe` accounts is `100`, then the limit for each `UserSafe` wallet would be `10e7/1000=100000`. The limit for each account (`1e5`) would be applied by only accounting for collateral up to the limit, that is, even if UserSafe contains > `1e5 ETH`, only `1e5` would be used as collateral.

Status: Acknowledged

Update from client: We don't think this would be the case because the tokens we would use are pegged to ETH/BTC or our liquid tokens.

6.0.3 [Low] Inconsistent updates of mode on swap

File(s): `UserSafeCore.sol`, `UserSafeSetters.sol`

Description: The modifier `currentMode(...)` is called on swap functions `swapAndSpend(...)` and `swapAndRepay(...)` however it is not called on `swap(...)` which creates inconsitent state updates on swap action.

Recommendation(s): Consider making consistent state updates on the swap.

Status: Acknowledged

Update from client: Design choice

6.0.4 [Low] Lack of partial liquidation

File(s): `DebtManagerCore`

Description: Current implementation allows only for a full liquidation of the UserSafe. This is not ideal for the users as in liquidation state the user could be completely liquidated. The partial liquidation allows for providing a more efficient system for protecting against bad debt but also protects users from losing all their funds.

Recommendation(s): Consider introducing partial liquidations.

Status: Fixed

Update from client: d56dea99969cac2210f0baa7d1ad0afa9c54794d

6.0.5 [Low] Liquidators can steal user's asset that is not part of the listed collateral tokens

File(s): `DebtManagerCore.sol`

Description: The protocol allows for liquidation of underwater positions. The liquidator is allowed to enter collateral tokens they would wish to liquidate with preferencially. The protocol then calls `_getCollateralTokensForDebtAmount()` to get the collateral assets from the user that is equivalent to the debt they would wish to offset.

The `_getCollateralTokensForDebtAmount()` checks the total amount of collateral per token a user has as `uint256 totalCollateral = IERC20(collateralToken).balanceOf(user);` then uses this to calculate the liquidation bonus the liquidator is supposed to get then at the end, send the calculated asset amount to the liquidator. However, if the passed in collateral tokens by the liquidator is not part of the collateral assets supported by the protocol, the liquidation will still happen and the user would lose an asset that was not even part of the liquidation process in the first place.

Consider the following example:

1. The supported collateral asset is only `wETH` and a user uses their `wETH` asset to get a loan in the protocol ;
2. The position gets underwater and is now ripe for liquidation ;
3. The user also has `wBTC` in their userSafe wallet ;
4. A malicious user would pass in `wBTC` as their preferred collateral token to get for liquidation and they would be able to get `wBTC` even though that asset was not even part of the loan position or listed as a collateral token. Basically, liquidators have the ability to steal any other token from the user that is not a collateral token ;

Recommendation(s): Consider sanitizing the user input collateral token to only be listed collateral tokens or in the userSafe, in `postLiquidate` perform the check on the asset to be transferred.

Status: Acknowledged, design choice.

Update from client: b0a44abca18112453393e383c00cc90f6bbf0f36

6.0.6 [Low] CashbackDispatcher::setCashbackToken allows setting cashback tokens that don't have the price configured yet

File(s): `CashbackDispatcher.sol`

Description: The 'setCashbackToken' function has two checks. First of all, it checks to see if the address of the new cashback token is 'address(0)', and then it checks if the price for the new '_token' was already configured or not.

```
    function setCashbackToken(address _token) external onlyRole(ADMIN_ROLE) {
        if (_token == address(0)) revert InvalidValue();
        if (priceProvider.price(cashbackToken) == 0) revert CashbackTokenPriceNotConfigured();
        emit CashbackTokenSet(cashbackToken, _token);
        cashbackToken = _token;
    }
```

The issue is that the function checks the price of the already existing ‘cashbackToken‘ instead of the new ‘_token‘ that’s being added. This allows the admin to set a new cashback ‘_token‘ where the price of the token is ‘0‘.

Recommendation(s): To properly check the price of the new ‘_token‘, this check ‘if (priceProvider.price(cashbackToken) == 0);‘ should be changed to this ‘if (priceProvider.price(_token) == 0);‘

Status: Fixed

Update from client: e55b1b5e0cb5f3d00a3844c773c3df3dd537e14b

6.0.7 [Low] Removal of the supported collateral may break the invariant

File(s): DebtManagerAdmin.sol

Description: One of the invariants in DebtManger is that every borrowing asset should also be a collateral asset. However, the `unsupportCollateralToken(...)` allows for unsupporting a collateral token while it is still a borrowing token. Such action will create a borrowing token that is not a collateral token.

Recommendation(s): Consider checking if the asset is not a borrowing token in the `unsupportCollateralToken(...)`.

Status: Fixed

Update from client: e55b1b5e0cb5f3d00a3844c773c3df3dd537e14b

6.0.8 [Low] Unsupporting a Borrow Token can be prevented through front-running

File(s): DebtManagerAdmin.sol

Description: For a token to be unsupported, the contract calls `_getTotalBorrowTokenAmount` to ensure there are no borrow positions opened on the borrow token as well as any borrow token in the contract. The `getTotalBorrowTokenAmount(...)` is defined as

```
function _getTotalBorrowTokenAmount(
      address borrowToken
  ) internal view returns (uint256) {
      return
          convertUsdToCollateralToken(borrowToken, totalBorrowingAmount(borrowToken)) +
          IERC20(borrowToken).balanceOf(address(this));
  }
```

Therefore, for a borrow token to be unsupported, there should be no borrow token present in the `debtManager` contract.

As a result, an attacker can frontrun `unsupportBorrowToken(...)` function to prevent borrow tokens from being unsupported.

Status: Acknowledged

Update from client: Acknowledged. Not important for us right now since we have only one spending token planned. In future if we plan to add more tokens, we can look into it.

6.0.9 [Low] Incorrect information from `canSpend(...)` and `maxCanSpend(...)` in UserSafe

File(s): DebtManagerAdmin.sol

Description: For a token to be unsupported, the contract calls `_getTotalBorrowTokenAmount` to ensure there are no borrow positions opened on the borrow token as well as any borrow token in the contract. The `getTotalBorrowTokenAmount(...)` is defined as

```
function _getTotalBorrowTokenAmount(
      address borrowToken
  ) internal view returns (uint256) {
      return
          convertUsdToCollateralToken(borrowToken, totalBorrowingAmount(borrowToken)) +
          IERC20(borrowToken).balanceOf(address(this));
  }
```

Therefore, for a borrow token to be unsupported, there should be no borrow token present in the `debtManager` contract.

As a result, an attacker can frontrun `unsupportBorrowToken(...)` function to prevent borrow tokens from being unsupported.

Recommendation(s): Consider adding the `block.timestamp > _incomingCreditModeStartTime` condition to the `canSpend(...)` and `maxCanSpend(...)` functions.

Status: Acknowledged

Update from client: Design choice, this is expected behavior.

6.0.10 [Info] Incorrect interface

File(s): `IL2DebtManager.sol`

Description: The `totalCollateralAmounts(...)` function was removed from DebtManager, but it's present in the interface.

Recommendation(s): Consider making the interface consistent with the DebtManger.

Status: Fixed

Update from client: 1d431d78856409aa0ed5a5325014c2ce421d4bf6

6.0.11 [Info] Unnecessary updates of mode `repay(...)` and `swapAndRepay(...)`

File(s): `UserSafeCore.sol`

Description: The `currentMode(...)` updates the mode and is called in `repay(...)` and `swapAndRepay(...)`. However, those actions do not involve spending funds. The mode is only relevent on the spend, so updating funds during the repay action is not necessary.

Recommendation(s): Consider reviewing if the call to `currentMode(...)` during repay is necessary and remove if not to save gas and keep state updates consistent.

Status: Fixed

Update from client: e55b1b5e0cb5f3d00a3844c773c3df3dd537e14b

e55b1b5e0cb5f3d00a3844c773c3df3dd537e14b

6.0.12 [Info] Unused internal function `_checkSpendingLimit(...)`

File(s): `UserSafeCore.sol`

Description: The `_checkSpendingLimit(...)` function is not used in the current implementation.

Recommendation(s): Consider removing unused function `_checkSpendingLimit(...)`.

Status: Fixed

Update from client: 1d431d78856409aa0ed5a5325014c2ce421d4bf6

6.0.13 [Info] Automatic switching to credit mode can return wrong possible expenditure for a user

File(s): `UserSafeCore.sol`

Description: The `canSpend (...)` view function returns the spenditure amount of a user. However, if a user has an incoming credit mode start date which has not yet materialized the function automatically switches the mode to credit mode and calculates prospective user's spenditure on credit mode and not debit mode. This is evident here `if (_incomingCreditModeStartTime != 0) __mode = Mode.Credit;`.

As a result, if a user wants to start a credit mode sometime in the future, it doesn't matter how far in the future, the `canSpend(...)` method will return spenditure amount based on credit mode and not the actual debit mode that is currently existing.

Recommendation(s): The protocol should consider using the existing current mode instead of the automatic switch.

Status: Acknowledged

Update from client: This is expected behavior and a design choice

6.0.14 [Info] The monthly spending limit is limited by the daily spending limit

File(s): `UserSafeCore.sol`

Description: The `SpendingLimitLib` library contract in calculating user's spending limit in the `spend(...)` function does so as:

```
if (limit.spentToday + amount > limit.dailyLimit) revert ExceededDailySpendingLimit();
if (limit.spentThisMonth + amount > limit.monthlyLimit) revert ExceededMonthlySpendingLimit();
```

Basically, the daily spend limit determines a monthly spend limit. So if a user for example has a daily spenditure limit of `$100` then their monthly limit will automatically be limited to `$3000` regardless of how high their monthly limit is.

Therefore, the monthly limit spend will always be limited by the set daily limit.

Recommendation(s):In the event there is a daily limit set, there is no need of setting the monthly limit

Status: Acknowledged

Update from client: This is expected behavior and a design choice

6.0.15 [Info] `UserSafeStorage::_swapFunds` will revert when working with `stETH` & `eETH`

File(s): `UserSafeCore.sol`

Description:The `_swapFunds()` function does not account for the 1-2 wei corner case of `stETH` and attempting to swap `stETH` or `eETH` is expected to always revert because of the assertion `if (IERC20(outputToken).balanceOf(address(this)) != balBefore + outputAmount) revert IncorrectOutputAmount();`

Under the hood `stETH` and `eETH` token transfers convert the tokens into shares, send the shares, and the receiver converts the shares back to tokens. Because of these 2 convertions and due to rounding down, the actual amount transferred will always be off by 1-2 wei. You can read more about it here

This affects the `UserSafeSetters::swap`, `UserSafeCore::swapAndSpend` and `UserSafeCore::swapAndRepay` functions.

Recommendation(s): Consider creating a separate assertion when the swapped tokens are `stETH`, `eETH` or other similar tokens. Keep in mind that fee-on-transfer token swaps will also fail because of this assertion.

Status: Acknowledged

Update from client: This does not affect us since the contracts will be deployed on Scroll which only has wrapped versions of these assets

6.0.16 [Info] credit Mode will not start at `_incomingCreditModeStartTime`

File(s):`UserSafeCore.sol`

Description: The `currentMode()` modifier checks if the `_incomingCreditModeStartTime` is set and if `block.timestamp > _incomingCreditModeStartTime` meaning if the set time to start the credit mode has passed it will set the mode to credit mode as `_mode = Mode.Credit`.

However, when the `_incomingCreditModeStartTime == block.timestamp`, the modifier would not update the mode to credit given the update only happens if the the `_incomingCreditModeStartTime` is higher than `block.timestamp`. As a result, although the start time for credit mode has reached it would not be set. The protocol should consider adding an = sign and refactor check to `_incomingCreditModeStartTime != 0 && block.timestamp > =_incomingCreditModeStartTime` to allow setting the mode to credit on the dot when the time has reached.

Recommendation(s): Consider updating the modifier `currentMode()` to `_incomingCreditModeStartTime != 0 && block.timestamp > =_incomingCreditModeStartTime`

Status: Acknowledged

Update from client: This is expected behavior and a design choice

6.0.17 [Best Practice] Add check if the token is supported on `repay(...)`

File(s): `DebtManagerCore.sol`

Description: The `repay(...)` function does not explicitly check if the provided token is supported as a borrow token. Repaying with other tokens is not possible because of the checks of value stored in the mapping '_userBorrowings', however, adding the explicit check would increase readability and provide better information for the user.

Recommendation(s): Consider checking if the token is supported as a borrowing token in the `repay(...)` function.

Status: Fixed

Update from client: e55b1b5e0cb5f3d00a3844c773c3df3dd537e14b

6.0.18 [Best Practice] Wrong code comments

File(s):`UserSafeCore.sol`

Description: The code comments here: here and here are written as `// user collateral for token in USD * 100 / liquidation threshold`. This is a misleading comment given the code is calculationg borrowable amount and using `ltv` and not liquidation threshold

Recommendation(s): Consider updating the code comments accordingly to accurately reflect the code

Status: Fixed.

Update from client: d70fa7f30f0972dd7ef4d43750a9796b35bcbc20

6.0.19 [Best Practice] Inconsistent use of 'ADMIN_ROLE' and 'DEFAULT_ADMIN_ROLE

File(s):`CashbackDispatcher.sol`

Description: The setter functions 'setCashDataProvider, setPriceProvider, and setCashbackToken' use the 'onlyRole(ADMIN_ROLE)' modifier, while the 'withdrawFunds and _authorizeUpgrade' functions use the 'onlyRole(DEFAULT_ADMIN_ROLE)' modifier.

This is redundant because the 'owner' of the contract has both roles assigned to it during initialization.

```
    function initialize(...) external initializer {
//..
//..
        __AccessControlDefaultAdminRules_init_unchained(5 * 60, _owner);
        _grantRole(ADMIN_ROLE, _owner);
//..
//..
    }
```

Recommendation(s): Consider using the ‘onlyRole(ADMIN_ROLE)‘ modifier across all functions for consistency.

Status: Acknowledged

Update from client: Design choice

6.0.20 [Best Practice] Lack of `address(0)` check for `recipient` param inside `requestWithdrawal` function

File(s): `UserSafeSetters`

Description: The `UserSafeCore::processWithdrawal` function will transfer the tokens to `_pendingWithdrawalRequest.recipient` address. The problem is that the `UserSafeSetters::requestWithdrawal` allows users to set `_pendingWithdrawalRequest.recipient` to `address(0)` leading to a potential loss of funds.

Recommendation(s): Although in order for this to happen the user must make a mistake, it can be easily prevented by adding a check `if (recipient == address(0)) revert();`.

Status: Fixed

Update from client: ed848d2e530949ee44ed60769eb73dc126bd9d5d

6.0.21 [Best Practice] No cap for the `borrowApy`

File(s): `DebtManagerAdmin.sol`

Description: The `_setBorrowApy` function only enforces that `apy != 0`, but there is no cap on the value of this variable. If the admin sets the `borrowApy` to a value that is too high, this can potentially lead to overflow during interest calculation.

Recommendation(s): Consider adding a new variable `MAX_BORROW_APY` into the contract, and check that the value of `apy` is between `0` and `MAX_BORROW_APY`. `if (apy == 0 || apy > MAX_BORROW_APY) revert InvalidValue();`

Status: Fixed

Update from client: b4a489cdd1673c6e7b4cb07a6464094bb311ba3f

7 Documentation Evaluation

Software documentation refers to the written or visual information that describes the functionality, architecture, design, and implementation of software. It provides a comprehensive overview of the software system and helps users, developers, and stakeholders understand how the software works, how to use it, and how to maintain it. Software documentation can take different forms, such as user manuals, system manuals, technical specifications, requirements documents, design documents, and code comments. Software documentation is critical in software development, enabling effective communication between developers, testers, users, and other stakeholders. It helps to ensure that everyone involved in the development process has a shared understanding of the software system and its functionality. Moreover, software documentation can improve software maintenance by providing a clear and complete understanding of the software system, making it easier for developers to maintain, modify, and update the software over time. Smart contracts can use various types of software documentation. Some of the most common types include:

- Technical whitepaper: A technical whitepaper is a comprehensive document describing the smart contract's design and technical details. It includes information about the purpose of the contract, its architecture, its components, and how they interact with each other;
- User manual: A user manual is a document that provides information about how to use the smart contract. It includes step-by-step instructions on how to perform various tasks and explains the different features and functionalities of the contract;
- Code documentation: Code documentation is a document that provides details about the code of the smart contract. It includes information about the functions, variables, and classes used in the code, as well as explanations of how they work;
- API documentation: API documentation is a document that provides information about the API (Application Programming Interface) of the smart contract. It includes details about the methods, parameters, and responses that can be used to interact with the contract;
- Testing documentation: Testing documentation is a document that provides information about how the smart contract was tested. It includes details about the test cases that were used, the results of the tests, and any issues that were identified during testing;
- Audit documentation: Audit documentation includes reports, notes, and other materials related to the security audit of the smart contract. This type of documentation is critical in ensuring that the smart contract is secure and free from vulnerabilities.

These types of documentation are essential for smart contract development and maintenance. They help ensure that the contract is properly designed, implemented, and tested, and they provide a reference for developers who need to modify or maintain the contract in the future.

Remarks about the EtherFi Cash documentation

The documentation for the EtherFi Cash protocol was provided by the team during kick-off meeting. Additional documentation in the form of diagrams.
The team answered every question during meetings or through messages, which gave the auditing team a lot of insight and a deep understanding of the technical aspects of the project.

8 Test Suite Evaluation

```
yarn run v1.22.22
$ forge test
[] Compiling...
No files changed, compilation skipped

Ran 1 test for test/Swapper/Swapper1Inch.t.sol:Swapper1InchV6Test
[PASS] test_Swap() (gas: 3179)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.68ms (24.13µs CPU time)

Ran 2 tests for test/AaveAdapter/AaveAdapter.t.sol:AaveAdapterTest
[PASS] test_AaveFullFlow() (gas: 129624)
[PASS] test_AaveProcess() (gas: 142579)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 2.28ms (184.08µs CPU time)

Ran 1 test for test/Swapper/SwapperOpenOcean.t.sol:SwapperOpenOceanTest
[PASS] test_Swap() (gas: 3156)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.10ms (24.17µs CPU time)

Ran 13 tests for test/CashDataProvider/CashDataProvider.t.sol:CashDataProviderTest
[PASS] test_Deploy() (gas: 87329)
[PASS] test_GrantEtherFiWalletRole() (gas: 84938)
[PASS] test_RevokeEtherFiWalletRole() (gas: 54115)
[PASS] test_SetCashbackDispatcher() (gas: 54251)
[PASS] test_SetDelay() (gas: 45571)
[PASS] test_SetEtherFiCashDebtManager() (gas: 54273)
[PASS] test_SetEtherFiCashMultiSig() (gas: 54366)
[PASS] test_SetPriceProvider() (gas: 54279)
[PASS] test_SetSwapper() (gas: 54355)
[PASS] test_SetUserSafeEventEmitter() (gas: 54340)
[PASS] test_SetUserSafeFactory() (gas: 54346)
[PASS] test_SetUserSafeLens() (gas: 54362)
[PASS] test_WhitelistUserSafe() (gas: 78609)
Suite result: ok. 13 passed; 0 failed; 0 skipped; finished in 2.96ms (929.79µs CPU time)

Ran 1 test for test/DebtManager/Deploy.t.sol:DebtManagerDeployTest
[PASS] test_Deploy() (gas: 200105)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 5.11ms (324.00µs CPU time)

Ran 6 tests for test/CashWrappedToken/CashWrappedToken.t.sol:CashWrappedTokenTest
[PASS] test_DeployCashWrappedToken() (gas: 30815)
[PASS] test_OnlyWhitelistedMintersCanMint() (gas: 339412)
[PASS] test_OnlyWhitelistedRecipientCanReceiveWrappedToken() (gas: 400768)
[PASS] test_WhitelistMinter() (gas: 72346)
[PASS] test_WhitelistRecipient() (gas: 72298)
[PASS] test_WithdrawWrappedToken() (gas: 359176)
Suite result: ok. 6 passed; 0 failed; 0 skipped; finished in 2.37ms (1.46ms CPU time)

Ran 3 tests for test/UserSafe/Deploy.t.sol:UserSafeDeployTest
[PASS] test_CannotDeployTwoUserSafesAtTheSameAddress() (gas: 1024198131)
[PASS] test_Deploy() (gas: 97950)
[PASS] test_DeployAUserSafe() (gas: 504982)
Suite result: ok. 3 passed; 0 failed; 0 skipped; finished in 6.23ms (681.13µs CPU time)

Ran 2 tests for test/CashWrappedToken/CashWrappedTokenFactory.t.sol:CashWrappedTokenFactoryTest
[PASS] test_DeployCashWrappedTokenFactory() (gas: 66477)
[PASS] test_Upgrade() (gas: 1722310)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 2.03ms (337.25µs CPU time)

Ran 8 tests for test/DebtManager/Collateral.t.sol:DebtManagerCollateralTest
[PASS] test_CanAddOrRemoveSupportedCollateralTokens() (gas: 930149)
[PASS] test_CannotAddCollateralTokenIfAlreadySupported() (gas: 30417)
[PASS] test_CannotAddNullAddressAsCollateralToken() (gas: 26084)
[PASS] test_CannotUnsupportAddressZeroAsCollateralToken() (gas: 23517)
[PASS] test_CannotUnsupportCollateralTokenWhichIsABorrowToken() (gas: 27846)
[PASS] test_CannotUnsupportTokenForCollateralIfItIsNotACollateralTokenAlready() (gas: 27967)
[PASS] test_LtvCannotBeGreaterThanLiquidationThreshold() (gas: 26388)
[PASS] test_OnlyAdminCanSupportOrUnsupportCollateral() (gas: 45305)
Suite result: ok. 8 passed; 0 failed; 0 skipped; finished in 8.03ms (758.17µs CPU time)
```

```
Ran 7 tests for test/DebtManager/SupplyAndWithdraw.t.sol:DebtManagerSupplyAndWithdrawTest
[PASS] test_CanOnlySupplyBorrowTokens() (gas: 22644)
[PASS] test_CannotWithdrawLessThanMinShares() (gas: 311415)
[PASS] test_CannotWithdrawTokenThatWasNotSupplied() (gas: 56497)
[PASS] test_SupplyAndWithdraw() (gas: 843239)
[PASS] test_SupplyEighteenDecimalsTwice() (gas: 654003)
[PASS] test_SupplyTwice() (gas: 511976)
[PASS] test_UserSafeCannotSupply() (gas: 35143)
Suite result: ok. 7 passed; 0 failed; 0 skipped; finished in 10.87ms (5.23ms CPU time)

Ran 11 tests for test/UserSafe/Spend.t.sol:UserSafeSpendTest
[PASS] test_CanSpendWithCreditModeEvenIfWithdrawalBlocksTheAmount() (gas: 1012212)
[PASS] test_CanSpendWithDebitModeEvenIfWithdrawalsBlockTheAmount() (gas: 594524)
[PASS] test_CannotSpendIfUnsupportedTokensForSpending() (gas: 71308)
[PASS] test_CannotSpendWithDebitFlowMoreThanSpendingLimit() (gas: 116539)
[PASS] test_CannotSpendWithDebitFlowWhenBalanceIsInsufficient() (gas: 253730)
[PASS] test_CannotSpendWithDebitModeIfBorrowExceedsMaxBorrow() (gas: 782889)
[PASS] test_CannotSpendWithSameTxIdTwice() (gas: 312329)
[PASS] test_CannotSwapAndSpendIfBalanceIsInsufficient() (gas: 65472)
[PASS] test_OnlyCashWalletCanSpendWithDebitFlow() (gas: 35788)
[PASS] test_SpendWithDebitFlow() (gas: 316115)
[PASS] test_SwapAndSpend() (gas: 557903)
Suite result: ok. 11 passed; 0 failed; 0 skipped; finished in 10.73ms (4.99ms CPU time)

Ran 6 tests for test/UserSafe/SpendingLimit.t.sol:UserSafeSpendingLimitTest
[PASS] test_CannotSpendMoreThanDailyOrMonthlySpendingLimit() (gas: 263608)
[PASS] test_DailyLimitCannotBeGreaterThanMonthlyLimit() (gas: 71742)
[PASS] test_DailySpendingLimitGetsRenewedAutomatically() (gas: 625617)
[PASS] test_UpdateDailySpendingLimitDoesNotDelayIfLimitIsGreater() (gas: 142570)
[PASS] test_UpdateMonthlySpendingLimitDoesNotDelayIfLimitIsGreater() (gas: 142619)
[PASS] test_UpdateSpendingLimit() (gas: 584847)
Suite result: ok. 6 passed; 0 failed; 0 skipped; finished in 11.19ms (3.19ms CPU time)

Ran 6 tests for test/DebtManager/Repay.t.sol:DebtManagerRepayTest
[PASS] test_CanRepayForOtherUser() (gas: 262577)
[PASS] test_CannotRepayWithNonBorrowToken() (gas: 45517)
[PASS] test_CannotRepayWithUsdcIfBalanceIsInsufficient() (gas: 222507)
[PASS] test_RepayAfterSomeTimeIncursInterestOnTheBorrowings() (gas: 311209)
[PASS] test_RepayWithUsdc() (gas: 300616)
[PASS] test_SwapAndRepay() (gas: 3157)
Suite result: ok. 6 passed; 0 failed; 0 skipped; finished in 6.18ms (1.46ms CPU time)

Ran 6 tests for test/DebtManager/Liquidate.t.sol:DebtManagerLiquidateTest
[PASS] test_CannotLiquidateIfNotLiquidatable() (gas: 165018)
[PASS] test_ChooseCollateralPreferenceWhenLiquidating() (gas: 1663042)
[PASS] test_Liquidate() (gas: 304911)
[PASS] test_LiquidatorIsChargedRightAmountOfBorrowTokens() (gas: 831923)
[PASS] test_OnlyAdminCanSetLiquidationThreshold() (gas: 31316)
[PASS] test_SetLiquidationThreshold() (gas: 38985)
Suite result: ok. 6 passed; 0 failed; 0 skipped; finished in 11.95ms (4.99ms CPU time)

Ran 1 test for test/IntegrationTest/IntegrationTest.t.sol:IntegrationTest
[PASS] test_MultipleSuppliers() (gas: 2538017)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 8.08ms (2.98ms CPU time)

Ran 2 tests for test/UserSafe/Owner.t.sol:UserSafeOwnerTest
[PASS] test_CanSetEthereumAddrAsOwner() (gas: 117266)
[PASS] test_CanSetPasskeyAsOwner() (gas: 141952)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 4.09ms (630.38µs CPU time)

Ran 2 tests for test/UserSafe/Swap.t.sol:UserSafeSwapTest
[PASS] test_CannotSwapIfBorrowPositionBecomesUnhealthy() (gas: 3111)
[PASS] test_Swap() (gas: 3156)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 4.68ms (21.38µs CPU time)
```

```
Ran 23 tests for test/DebtManager/Borrow.t.sol:DebtManagerBorrowTest
[PASS] test_Borrow() (gas: 488677)
[PASS] test_BorrowApyCannotBeZero() (gas: 25837)
[PASS] test_BorrowIncursInterestWithTime() (gas: 431857)
[PASS] test_BorrowTokenMinSharesCannotBeZero() (gas: 25817)
[PASS] test_BorrowTokenWithDecimalsOtherThanSix() (gas: 1321774)
[PASS] test_CanAddOrRemoveSupportedBorrowTokens() (gas: 776962)
[PASS] test_CanSetBorrowApy() (gas: 77296)
[PASS] test_CanSetMinBorrowTokenShares() (gas: 76933)
[PASS] test_CannotAddBorrowTokenIfAlreadySupported() (gas: 30312)
[PASS] test_CannotBorrowIfDebtRatioGreaterThanThreshold() (gas: 524441)
[PASS] test_CannotBorrowIfNoCollateral() (gas: 482800)
[PASS] test_CannotBorrowIfNotUserSafe() (gas: 28059)
[PASS] test_CannotBorrowIfTokenIsNotSupported() (gas: 32737)
[PASS] test_CannotBorrowIfUsdcBalanceInsufficientInDebtManager() (gas: 522629)
[PASS] test_CannotRemoveSupportIfBorrowTokenIsStillInTheSystem() (gas: 64356)
[PASS] test_CannotSetBorrowApyForUnsupportedToken() (gas: 28031)
[PASS] test_CannotSetBorrowTokenMinSharesForUnsupportedToken() (gas: 28016)
[PASS] test_CannotUnsupportAllTokensAsBorrowTokens() (gas: 147638)
[PASS] test_CannotUnsupportTokenForBorrowIfItIsNotABorrowTokenAlready() (gas: 27891)
[PASS] test_NextBorrowAutomaticallyAddsInterestToThePreviousBorrows() (gas: 535120)
[PASS] test_OnlyAdminCanSetBorrowApy() (gas: 26571)
[PASS] test_OnlyAdminCanSetBorrowTokenMinShares() (gas: 26612)
[PASS] test_OnlyAdminCanSupportOrUnsupportBorrowTokens() (gas: 630567)
Suite result: ok. 23 passed; 0 failed; 0 skipped; finished in 7.43ms (3.41ms CPU time)

Ran 24 tests for test/TopUp/TopUpDest.t.sol:TopUpDestTest
[PASS] test_CanBatchTopUpUserSafe() (gas: 407516)
[PASS] test_CanDeposit() (gas: 110367)
[PASS] test_CanRegisterWalletToUserSafe() (gas: 99169)
[PASS] test_CanTopUpUserSafe() (gas: 141925)
[PASS] test_CanWithdraw() (gas: 88657)
[PASS] test_CannotBatchTopUpUserSafeIfArrayLengthMismatch() (gas: 82121)
[PASS] test_CannotDepositZeroAmount() (gas: 20691)
[PASS] test_CannotPauseIfAlreadyPaused() (gas: 43391)
[PASS] test_CannotRegisterWalletToUserSafeIfDeadlineHasPassed() (gas: 29289)
[PASS] test_CannotRegisterWalletToUserSafeIfSignatureIsInvalid() (gas: 72737)
[PASS] test_CannotRegisterWalletToUserSafeIfUserSafeIsNotRegistered() (gas: 26786)
[PASS] test_CannotRegisterWalletToUserSafeIfWalletIsAddressZero() (gas: 14327)
[PASS] test_CannotTopUpIfBalanceTooLow() (gas: 65959)
[PASS] test_CannotTopUpIfNotAValidUserSafe() (gas: 106989)
[PASS] test_CannotTopUpIfPaused() (gas: 47649)
[PASS] test_CannotTopUpIfTxIdAlreadyCompleted() (gas: 141983)
[PASS] test_CannotUnpauseIfNotPaused() (gas: 20167)
[PASS] test_CannotWithdrawIfAmountGreaterThanDeposit() (gas: 23237)
[PASS] test_CannotWithdrawIfBalanceTooLow() (gas: 142082)
[PASS] test_CannotWithdrawZeroAmount() (gas: 20968)
[PASS] test_OnlyDefaultAdminCanUnPause() (gas: 62924)
[PASS] test_OnlyDepositorCanDeposit() (gas: 208729)
[PASS] test_OnlyPauserCanPause() (gas: 73163)
[PASS] test_OnlyTopUpRoleCanTopUpUserSafe() (gas: 96959)
Suite result: ok. 24 passed; 0 failed; 0 skipped; finished in 3.09ms (1.86ms CPU time)

Ran 5 tests for test/UserSafe/Mode.t.sol:UserSafeModeTest
[PASS] test_CanSetDebitModeIfBorrowIsNotRepaid() (gas: 750722)
[PASS] test_CannotSetTheSameMode() (gas: 169285)
[PASS] test_InitialModeIsDebit() (gas: 20673)
[PASS] test_SwitchToCreditModeIncursDelay() (gas: 127940)
[PASS] test_SwitchToDebitModeDoesNotIncursDelay() (gas: 127524)
Suite result: ok. 5 passed; 0 failed; 0 skipped; finished in 8.65ms (3.15ms CPU time)

Ran 4 tests for test/UserSafe/PendingCashback.t.sol:UserSafePendingCashbackTest
[PASS] test_CanGetPendingCashback() (gas: 472811)
[PASS] test_RetrievePendingCashback() (gas: 655757)
[PASS] test_RetrievePendingCashbackWhenBalNotAvailableJustReturns() (gas: 488804)
[PASS] test_RetrievePendingCashbackWhenNoPendingCashbackJustReturns() (gas: 29370)
Suite result: ok. 4 passed; 0 failed; 0 skipped; finished in 6.18ms (2.76ms CPU time)
```

```
Ran 9 tests for test/UserSafe/Recovery.t.sol:UserSafeRecoveryTest
[PASS] test_CanRecoverWithTwoAuthorizedSignatures() (gas: 455977)
[PASS] test_CanSetIsRecoveryActive() (gas: 97177)
[PASS] test_CannotRecoverIfRecoveryIndexIsInvalid() (gas: 131423)
[PASS] test_CannotRecoverIfRecoveryIsInactive() (gas: 157929)
[PASS] test_IsRecoveryActive() (gas: 18376)
[PASS] test_RecoveryFailsIfRecoveryOwnerIsNotSetAndIndexPassedIsZero() (gas: 64897)
[PASS] test_RecoveryFailsIfSignatureIndicesAreSame() (gas: 59918)
[PASS] test_RecoveryFailsIfSignatureIsInvalid() (gas: 143662)
[PASS] test_UserCanCancelRecoveryIfMaliciousRecovery() (gas: 212359)
Suite result: ok. 9 passed; 0 failed; 0 skipped; finished in 10.82ms (8.57ms CPU time)

Ran 3 tests for test/UserSafe/UserSafeLens.t.sol:UserSafeLensTest
[PASS] test_CanGetUserDataWithBorrows() (gas: 880325)
[PASS] test_CanGetUserDataWithWithdrawals() (gas: 432721)
[PASS] test_Deploy() (gas: 231944)
Suite result: ok. 3 passed; 0 failed; 0 skipped; finished in 8.90ms (3.65ms CPU time)

Ran 12 tests for test/UserSafe/UserSafeFactory.t.sol:UserSafeFactoryTest
[PASS] test_CannotSetBeaconToAddressZero() (gas: 18090)
[PASS] test_CannotSetCashDataProviderToAddressZero() (gas: 18180)
[PASS] test_Deploy() (gas: 69896)
[PASS] test_OnlyAdminCanCreateUserSafe() (gas: 29478)
[PASS] test_OnlyAdminCanSetBeacon() (gas: 18905)
[PASS] test_OnlyAdminCanSetCashDataProvider() (gas: 18975)
[PASS] test_OnlyOwnerCanUpgradeFactoryImpl() (gas: 19644)
[PASS] test_OnlyOwnerCanUpgradeUserSafeImpl() (gas: 21125)
[PASS] test_SetBeacon() (gas: 29423)
[PASS] test_SetCashDataProvider() (gas: 29461)
[PASS] test_UpgradeUserSafeFactory() (gas: 2742316)
[PASS] test_UpgradeUserSafeImpl() (gas: 49284)
Suite result: ok. 12 passed; 0 failed; 0 skipped; finished in 5.16ms (329.46µs CPU time)

Ran 2 tests for test/UserSafe/WebAuthn.t.sol:UserSafeWebAuthnSignatureTest
[PASS] test_CanSetOwnerWithWebAuthn() (gas: 346178)
[PASS] test_Deploy() (gas: 95359)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 4.96ms (1.67ms CPU time)

Ran 10 tests for test/UserSafe/Withdrawal.t.sol:UserSafeWithdrawalTest
[PASS] test_CanResetWithdrawalWithNewRequest() (gas: 490152)
[PASS] test_CanTransferEvenIfAmountIsBlockedByWithdrawal() (gas: 523706)
[PASS] test_CannotProcessWithdrawalsBeforeTime() (gas: 416155)
[PASS] test_CannotRequestWithdrawalIfArrayLengthMismatch() (gas: 81099)
[PASS] test_CannotRequestWithdrawalWhenFundsAreInsufficient() (gas: 124858)
[PASS] test_CannotRequestWithdrawalWhenRecipientIsAddressZero() (gas: 393556)
[PASS] test_CannotRequestWithdrawalWithDuplicateTokens() (gas: 66017)
[PASS] test_CannotWithdrawIfAccountBecomesUnhealthy() (gas: 1058387)
[PASS] test_ProcessWithdrawals() (gas: 388655)
[PASS] test_RequestWithdrawalWithPermit() (gas: 432872)
Suite result: ok. 10 passed; 0 failed; 0 skipped; finished in 5.77ms (5.96ms CPU time)

Ran 9 tests for test/TimeLib.t.sol:TimeLibTest
[PASS] test_FuzzGetStartOfNextDay(uint256,int256) (runs: 256, : 6415, ~: 6415)
[PASS] test_FuzzGetStartOfNextMonth(uint256,int256) (runs: 256, : 12608, ~: 12608)
[PASS] test_FuzzGetStartOfNextWeek(uint256,int256) (runs: 256, : 6650, ~: 6651)
[PASS] test_GetStartOfNextDay() (gas: 8028)
[PASS] test_GetStartOfNextDay_LeapYear() (gas: 5533)
[PASS] test_GetStartOfNextMonth() (gas: 40625)
[PASS] test_GetStartOfNextMonth_LeapYear() (gas: 17853)
[PASS] test_GetStartOfNextWeek() (gas: 10660)
[PASS] test_GetStartOfNextWeek_LeapYear() (gas: 5944)
Suite result: ok. 9 passed; 0 failed; 0 skipped; finished in 35.08ms (37.03ms CPU time)
```

```
Ran 19 tests for test/UserSafe/CanSpend.t.sol:UserSafeCanSpendTest
[PASS] test_CanSpendFailsIfTxIdIsAlreadyCleared() (gas: 424653)
[PASS] test_CanSpendWithCreditModeFailsIfCollateralBalanceIsZero() (gas: 196920)
[PASS] test_CanSpendWithCreditModeFailsIfWithdrawalRequestBlocksIt() (gas: 502602)
[PASS] test_CanSpendWithCreditModeIfAfterWithdrawalAmountIsStillBorrowable() (gas: 647722)
[PASS] test_CanSpendWithCreditModeIfBalAvailable() (gas: 518082)
[PASS] test_CanSpendWithDebitModeFailsIfBalTooLow() (gas: 227301)
[PASS] test_CanSpendWithDebitModeFailsIfDailyLimitIsLowerThanAmountUsed() (gas: 550570)
[PASS] test_CanSpendWithDebitModeFailsIfDailySpendingLimitIsExhausted() (gas: 459138)
[PASS] test_CanSpendWithDebitModeFailsIfDailySpendingLimitIsTooLow() (gas: 379839)
[PASS] test_CanSpendWithDebitModeFailsIfIncomingDailyLimitIsLowerThanAmountUsed() (gas: 547950)
[PASS] test_CanSpendWithDebitModeFailsIfIncomingDailySpendingLimitIsExhausted() (gas: 571551)
[PASS] test_CanSpendWithDebitModeFailsIfIncomingDailySpendingLimitIsTooLow() (gas: 377464)
[PASS] test_CanSpendWithDebitModeFailsIfWithdrawalRequestBlocksIt() (gas: 450556)
[PASS] test_CanSpendWithDebitModeFailsIfWithdrawalRequestBlocksIt2() (gas: 638687)
[PASS] test_CanSpendWithDebitModeIfBalAvailable() (gas: 275622)
[PASS] test_CanSpendWithDebitModeIfIncomingDailySpendingLimitRenews() (gas: 536024)
[PASS] test_CanSpendWithDebitModeIfSpendingLimitRenews() (gas: 464290)
[PASS] test_CanSpendWithDebitModeIfWithdrawalIsLowerThanAmountRequested() (gas: 478176)
[PASS] test_CannotSpendWithCreditModeIfLiquidityAvailableInDebtManager() (gas: 354194)
Suite result: ok. 19 passed; 0 failed; 0 skipped; finished in 19.04ms (11.10ms CPU time)

Ran 6 tests for test/PriceProvider/PriceProvider.t.sol:PriceProviderTest
[PASS] test_BtcOracle() (gas: 87341)
[PASS] test_CanAddNewOracle() (gas: 123996)
[PASS] test_CannotAddNewOracleIfArrayLengthMismatch() (gas: 32219)
[PASS] test_ExchangeRatePrice() (gas: 102967)
[PASS] test_MaticUsdOracle() (gas: 123983)
[PASS] test_OnlyOwnerCanAddNewOracle() (gas: 33226)
Suite result: ok. 6 passed; 0 failed; 0 skipped; finished in 8.37s (6.47s CPU time)

Ran 18 tests for test/SettlementDispatcher/SettlementDispatcher.t.sol:CashSafeTest
[PASS] test_Bridge() (gas: 337267)
[PASS] test_CanSetDestinationData() (gas: 49158)
[PASS] test_CannotBridgeFundsIfBalanceInsufficient() (gas: 35385)
[PASS] test_CannotBridgeFundsIfNoFee() (gas: 246904)
[PASS] test_CannotBridgeIfDestDataIsNotSet() (gas: 191058)
[PASS] test_CannotBridgeIfInvalidValuesArePassed() (gas: 29724)
[PASS] test_CannotBridgeIfMinReturnIsTooLow() (gas: 246951)
[PASS] test_CannotSetDestDataIfArrayLengthMismatch() (gas: 19640)
[PASS] test_CannotSetInvalidStargateValue() (gas: 33417)
[PASS] test_CannotSetInvalidValuesInDestData() (gas: 50236)
[PASS] test_CannotWithdrawIfInsufficientBalance() (gas: 85776)
[PASS] test_CannotWithdrawIfNoBalance() (gas: 44447)
[PASS] test_CannotWithdrawIfRecipientIsAddressZero() (gas: 20431)
[PASS] test_Deploy() (gas: 56427)
[PASS] test_OnlyAdminCanSetDestData() (gas: 27058)
[PASS] test_OnlyBridgerCanBridgeFunds() (gas: 21069)
[PASS] test_WithdrawErc20Funds() (gas: 240103)
[PASS] test_WithdrawNativeFunds() (gas: 65036)
Suite result: ok. 18 passed; 0 failed; 0 skipped; finished in 8.97s (22.31s CPU time)

Ran 11 tests for test/PreOrder.t.sol:PreOrderTest
[PASS] testAssemblyProperlySetsArrayLength() (gas: 19783)
[PASS] testMint() (gas: 4811368)
[PASS] testMintWithPermit() (gas: 3250356)
[PASS] testPause() (gas: 115710)
[PASS] testRevert() (gas: 2990366)
[PASS] testSellOutTiers() (gas: 3429236)
[PASS] testTiersLengthCheck() (gas: 15806997)
[PASS] testTokensForUser() (gas: 33308018)
[PASS] testURI() (gas: 37764)
[PASS] test_CanMintWithFiat() (gas: 56993)
[PASS] test_OnlyFiatMinterCanMintWithFiat() (gas: 21095)
Suite result: ok. 11 passed; 0 failed; 0 skipped; finished in 10.99s (11.01s CPU time)
```

```
Ran 27 tests for test/CashbackDispatcher/CashbackDispatcher.t.sol:CashbackDispatcherTest
[PASS] test_CanSetCashDataProvider() (gas: 28184)
[PASS] test_CanSetCashbackToken() (gas: 37963)
[PASS] test_CanSetPriceProvider() (gas: 33022)
[PASS] test_CannotSetAddressZeroAsCashDataProvider() (gas: 18204)
[PASS] test_CannotSetAddressZeroAsCashbackToken() (gas: 18229)
[PASS] test_CannotSetAddressZeroAsPriceProvider() (gas: 18174)
[PASS] test_CannotWithdrawIfInsufficientBalance() (gas: 72136)
[PASS] test_CannotWithdrawIfNoBalance() (gas: 37380)
[PASS] test_CannotWithdrawIfRecipientIsAddressZero() (gas: 20459)
[PASS] test_CashbackPaidCreditFlowChad() (gas: 931592)
[PASS] test_CashbackPaidCreditFlowPepe() (gas: 921382)
[PASS] test_CashbackPaidCreditFlowWhale() (gas: 931590)
[PASS] test_CashbackPaidCreditFlowWojak() (gas: 931573)
[PASS] test_CashbackPaidDebitFlowChad() (gas: 626625)
[PASS] test_CashbackPaidDebitFlowPepe() (gas: 616488)
[PASS] test_CashbackPaidDebitFlowWhale() (gas: 626627)
[PASS] test_CashbackPaidDebitFlowWojak() (gas: 626624)
[PASS] test_CashbackUnpaid() (gas: 475445)
[PASS] test_Deploy() (gas: 69440)
[PASS] test_OnlyOwnerCanSetCashDataProvider() (gas: 22482)
[PASS] test_OnlyOwnerCanSetCashbackToken() (gas: 22505)
[PASS] test_OnlyOwnerCanSetPriceProvider() (gas: 22495)
[PASS] test_UnpaidCashbackIsAccumulatedInNextTx() (gas: 619408)
[PASS] test_UnpaidCashbackIsPaidAndAccumulatesIfTotalBalNotAvailableInNextTx() (gas: 825027)
[PASS] test_UnpaidCashbackIsPaidIfBalAvailableInNextTx() (gas: 811923)
[PASS] test_WithdrawErc20Funds() (gas: 207801)
[PASS] test_WithdrawNativeFunds() (gas: 65094)
Suite result: ok. 27 passed; 0 failed; 0 skipped; finished in 11.00s (12.94ms CPU time)

Ran 2 tests for test/AaveUnitTest.t.sol:AaveUnitTest
[PASS] test_fullFlow() (gas: 805966)
[PASS] test_healthFactor() (gas: 486079)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 33.59s (52.03s CPU time)

Ran 24 tests for test/TopUp/TopUpSource.t.sol:TopUpSourceTest
[PASS] test_BridgeUsdc() (gas: 345848)
[PASS] test_BridgeWeETH() (gas: 580293)
[PASS] test_CanSetRecoveryWallet() (gas: 31923)
[PASS] test_CanSetTokenConfig() (gas: 114930)
[PASS] test_CannotBridgeIfInsufficientNativeFee() (gas: 262508)
[PASS] test_CannotBridgeIfTokenBalanceIsZero() (gas: 37686)
[PASS] test_CannotBridgeIfTokenConfigNotSet() (gas: 24926)
[PASS] test_CannotBridgeIfTokenIsAddressZero() (gas: 20638)
[PASS] test_CannotBridgeWhenPaused() (gas: 45350)
[PASS] test_CannotPauseIfAlreadyPaused() (gas: 43105)
[PASS] test_CannotRecoverIfTokenIsAddressZero() (gas: 20343)
[PASS] test_CannotSetRecoveryWalletToAddressZero() (gas: 18518)
[PASS] test_CannotSetTokenConfigForNullToken() (gas: 26753)
[PASS] test_CannotSetTokenConfigIfArrayLengthMismatch() (gas: 20436)
[PASS] test_CannotSetTokenConfigIfInvalidConfig() (gas: 54144)
[PASS] test_CannotUnpauseIfNotPaused() (gas: 17779)
[PASS] test_Deploy() (gas: 46641)
[PASS] test_EthConvertsToWeth() (gas: 62204)
[PASS] test_OnlyDefaultAdminCanRecoverFunds() (gas: 22505)
[PASS] test_OnlyDefaultAdminCanUnPause() (gas: 63362)
[PASS] test_OnlyOwnerCanSetRecoveryWallet() (gas: 20403)
[PASS] test_OnlyOwnerCanSetTokenConfig() (gas: 30859)
[PASS] test_OnlyPauserCanPause() (gas: 73361)
[PASS] test_Recovery() (gas: 391710)
Suite result: ok. 24 passed; 0 failed; 0 skipped; finished in 35.05s (35.06s CPU time)

Ran 34 test suites in 35.06s (108.18s CPU time): 286 tests passed, 0 failed, 0 skipped (286 total tests)
  Done in 35.98s.
```

9 About Nethermind

Nethermind is a Blockchain Research and Software Engineering company. Our work touches every part of the web3 ecosystem - from layer 1 and layer 2 engineering, cryptography research, and security to application-layer protocol development. We offer strategic support to our institutional and enterprise partners across the blockchain, digital assets, and DeFi sectors, guiding them through all stages of the research and development process, from initial concepts to successful implementation.

We offer security audits of projects built on EVM-compatible chains and Starknet. We are active builders of the Starknet ecosystem, delivering a node implementation, a block explorer, a Solidity-to-Cairo transpiler, and formal verification tooling. Nethermind also provides strategic support to our institutional and enterprise partners in blockchain, digital assets, and decentralized finance (DeFi). In the next paragraphs, we introduce the company in more detail.

Blockchain Security: At Nethermind, we believe security is vital to the health and longevity of the entire Web3 ecosystem. We provide security services related to Smart Contract Audits, Formal Verification, and Real-Time Monitoring. Our Security Team comprises blockchain security experts in each field, often collaborating to produce comprehensive and robust security solutions. The team has a strong academic background, can apply state-of-the-art techniques, and is experienced in analyzing cutting-edge Solidity and Cairo smart contracts, such as ArgentX and StarkGate (the bridge connecting Ethereum and StarkNet). Most team members hold a Ph.D. degree and actively participate in the research community, accounting for 240+ articles published and 1,450+ citations in Google Scholar. The security team adopts customer-oriented and interactive processes where clients are involved in all stages of the work.

Blockchain Core Development: Our core engineering team, consisting of over 20 developers, maintains, improves, and upgrades our flagship product - the Nethermind Ethereum Execution Client. The client has been successfully operating for several years, supporting both the Ethereum Mainnet and its testnets, and now accounts for nearly a quarter of all synced Mainnet nodes. Our unwavering commitment to Ethereum's growth and stability extends to sidechains and layer 2 solutions. Notably, we were the sole execution layer client to facilitate Gnosis Chain's Merge, transitioning from Aura to Proof of Stake (PoS), and we are actively developing a full-node client to bolster Starknet's decentralization efforts. Our core team equips partners with tools for seamless node set-up, using generated docker-compose scripts tailored to their chosen execution client and preferred configurations for various network types.

DevOps and Infrastructure Management: Our infrastructure team ensures our partners' systems operate securely, reliably, and efficiently. We provide infrastructure design, deployment, monitoring, maintenance, and troubleshooting support, allowing you to focus on your core business operations. Boasting extensive expertise in Blockchain as a Service, private blockchain implementations, and node management, our infrastructure and DevOps engineers are proficient with major cloud solution providers and can host applications in-house or on clients' premises. Our global in-house SRE teams offer 24/7 monitoring and alerts for both infrastructure and application levels. We manage over 5,000 public and private validators and maintain nodes on major public blockchains such as Polygon, Gnosis, Solana, Cosmos, Near, Avalanche, Polkadot, Aptos, and StarkWare L2. Sedge is an open-source tool developed by our infrastructure experts, designed to simplify the complex process of setting up a proof-of-stake (PoS) network or chain validator. Sedge generates docker-compose scripts for the entire validator set-up based on the chosen client, making the process easier and quicker while following best practices to avoid downtime and being slashed.

Cryptography Research: At Nethermind, our Cryptography Research team is dedicated to continuous internal research while fostering close collaboration with external partners. The team has expertise across a wide range of domains, including cryptography protocols, consensus design, decentralized identity, verifiable credentials, Sybil resistance, oracles, and credentials, distributed validator technology (DVT), and Zero-knowledge proofs. This diverse skill set, combined with strong collaboration between our engineering teams, enables us to deliver cutting-edge solutions to our partners and clients.

Smart Contract Development & DeFi Research: Our smart contract development and DeFi research team comprises 40+ world-class engineers who collaborate closely with partners to identify needs and work on value-adding projects. The team specializes in Solidity and Cairo development, architecture design, and DeFi solutions, including DEXs, AMMs, structured products, derivatives, and money market protocols, as well as ERC20, 721, and 1155 token design. Our research and data analytics focuses on three key areas: technical due diligence, market research, and DeFi research. Utilizing a data-driven approach, we offer in-depth insights and outlooks on various industry themes.

Our suite of L2 tooling: Warp is Starknet's approach to EVM compatibility. It allows developers to take their Solidity smart contracts and transpile them to Cairo, Starknet's smart contract language. In the short time since its inception, the project has accomplished many achievements, including successfully transpiling Uniswap v3 onto Starknet using Warp.

- **Voyager** is a user-friendly Starknet block explorer that offers comprehensive insights into the Starknet network. With its intuitive interface and powerful features, Voyager allows users to easily search for and examine transactions, addresses, and contract details. As an essential tool for navigating the Starknet ecosystem, Voyager is the go-to solution for users seeking in-depth information and analysis;
- **Horus** is an open-source formal verification tool for StarkNet smart contracts. It simplifies the process of formally verifying Starknet smart contracts, allowing developers to express various assertions about the behavior of their code using a simple assertion language;
- **Juno** is a full-node client implementation for Starknet, drawing on the expertise gained from developing the Nethermind Client. Written in Golang and open-sourced from the outset, Juno verifies the validity of the data received from Starknet by comparing it to proofs retrieved from Ethereum, thus maintaining the integrity and security of the entire ecosystem.

Learn more about us at nethermind.io.

General Advisory to Clients

As auditors, we recommend that any changes or updates made to the audited codebase undergo a re-audit or security review to address potential vulnerabilities or risks introduced by the modifications. By conducting a re-audit or security review of the modified codebase, you can significantly enhance the overall security of your system and reduce the likelihood of exploitation. However, we do not possess the authority or right to impose obligations or restrictions on our clients regarding codebase updates, modifications, or subsequent audits. Accordingly, the decision to seek a re-audit or security review lies solely with you.

Disclaimer

This report is based on the scope of materials and documentation provided by you to Nethermind in order that Nethermind could conduct the security review outlined in **1. Executive Summary** and **2. Audited Files**. The results set out in this report may not be complete nor inclusive of all vulnerabilities. Nethermind has provided the review and this report on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. Blockchain technology remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. This report does not indicate the endorsement of any particular project or team, nor guarantee its security. No third party should rely on this report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. To the fullest extent permitted by law, Nethermind disclaims any liability in connection with this report, its content, and any related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Nethermind does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and Nethermind will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.